

浙江大学

本科实验报告

课程名称: 操作系统

姓 名:

学 院: 计算机科学与技术学院

系: 计算机科学与技术系

专 业: 计算机科学与技术

学 号:

指导教师: 李环

2024 年 11 月 4 日

浙江大学操作系统实验报告

实验名称：_____RV64 虚拟内存管理_____

电子邮件地址：_____手机：_____

实验地点：_____曹西 503_____实验日期：2024 年 11 月 4 日

一、实验目的和要求

学习虚拟内存的相关知识，实现物理地址到虚拟地址的切换

了解 RISC-V 架构中 SV39 分页模式，实现虚拟地址到物理地址的映射，并对不同的段进行相应的权限设置

二、实验过程

（一）准备工程

在 defs.h 添加如下内容：

```
1.  #define OPENSBI_SIZE (0x200000)
2.
3.  #define VM_START (0xffffffe000000000)
4.  #define VM_END (0xfffffffff00000000)
5.  #define VM_SIZE (VM_END - VM_START)
6.
7.  #define PA2VA_OFFSET (VM_START - PHY_START)
```

从 repo 同步 vmlinux.lds 代码

在开始实验开启虚拟地址之前，我们还需要对 Makefile 进行一些修改来防止后面运行 / 调试出现问题：

```
1.  CF = -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin
    -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc
    -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g
```

（二）开启虚拟内存映射

在 RISC-V 中开启虚拟地址被分为了两步：setup_vm 以及 setup_vm_final，下面将介绍相关的具体实现。

1. setup_vm()的实现

将 0x80000000 开始的 1GB 区域进行两次映射，其中一次是等值映射（PA == VA），另一次是将其映射到 direct mapping area（使得 $PA + PV2VA_OFFSET == VA$ ），如下图所示：



完成上述映射之后，通过 relocate 函数，完成对 satp 的设置，以及跳转到对应的虚拟地址至此我们已经完成了虚拟地址的开启，之后我们运行的代码也都将在虚拟地址上运行

```
1.  uint64_t early_ptbl[512] __attribute__((__aligned__(0x1000)));
2.
3.  void setup_vm() {
4.      /*
5.       * 1. 由于是进行 1GiB 的映射，这里不需要使用多级页表
6.       * 2. 将 va 的 64bit 作为如下划分： | high bit | 9 bit | 30 bit |
7.       *     high bit 可以忽略
8.       *     中间 9 bit 作为 early_ptbl 的 index
9.       *     低 30 bit 作为页内偏移，这里注意到 30 = 9 + 9 + 12，即我们只使用根页表，根页表的每个 entry 都对应 1GiB 的区域
10.      * 3. Page Table Entry 的权限 V | R | W | X 位设置为 1
11.      */
12.      memset(early_ptbl, 0, PGSIZE);
```

```

13.     early_pgtbl[PHY_START >> 30 & 0x1ff] = (PHY_START >> 30 & 0x3f
        fffff) << 28 | 15;
14.     early_pgtbl[VM_START >> 30 & 0x1ff] = (PHY_START >> 30 & 0x3ff
        ffff) << 28 | 15;
15.     return;
16. }

```

```

1.  relocate:
2.      li t0, PA2VA_OFFSET
3.      add ra, ra, t0
4.      add sp, sp, t0
5.
6.      la t2, early_pgtbl
7.      sub t2, t2, t0
8.      srli t2, t2, 12
9.      addi t0, x0, 1
10.     li t1, 63
11.     sll t0, t0, t1
12.     or t2, t2, t0
13.
14.     sfence.vma zero, zero
15.     csrw satp, t2
16.
17.
18.     ret

```

2. setup_vm_final() 的实现

经过 `setup_vm` 设置了一级页表之后，整个 `kernel` 启动后应该可以直接运行在虚拟地址上了，不过在 `task_init` 中 `kalloc` 的时候可能会出现错误，因为 `mm_init` 中我们释放的内存是 `_ekernel ~ PHY_END`，在进入虚拟地址后 `PHY_END` 还是物理地址，会导致实际上没有内存被释放，`kalloc` 没有可用的空间。因此需要修改 `arch/riscv/kernel/mm.c` 的 `mm_init` 函数，将结束地址调整为虚拟地址，才能正常运行。

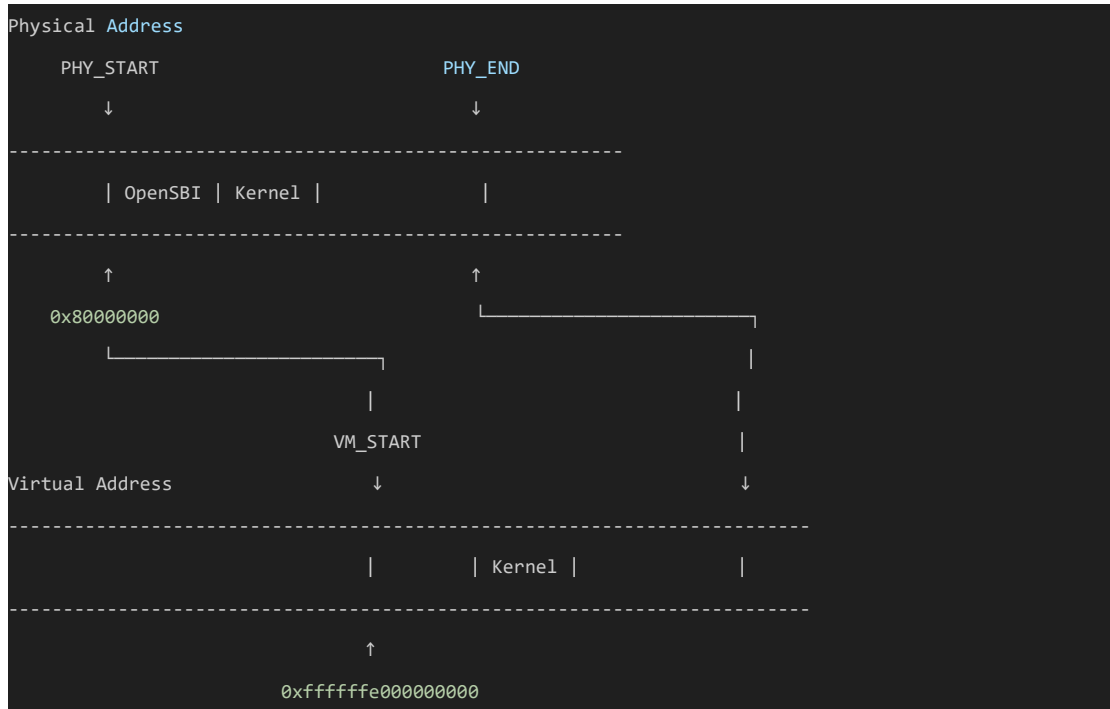
```

1.  void mm_init(void) {
2.      kfreerange(_ekernel, (char *) (PHY_END + PA2VA_OFFSET));
3.      printk("...mm_init done!\n");
4.  }

```

由于 `setup_vm_final` 中需要申请页面来建立多级页表，我们需要先调用 `mm_init` 来完成内存管理初始化，接下来 `setup_vm_final` 需要完成对所有物理内存 (128M) 的映射，并设置

正确的权限，具体的映射关系如下：



此时不再需要进行等值映射，不再需要为 OpenSBI 创建映射，因为 OpenSBI 运行在 M 态，直接使用物理地址，并且采用三级页表映射。

在 head.S 中 适当的位置调用 setup_vm_final

```
1.  uint64_t swapper_pg_dir[512] __attribute__((__aligned__(0x1000)))  
   ;  
2.  
3.  extern uint64_t _skernel, _stext, _srodata, _sdata, _sbss;  
4.  void setup_vm_final() {  
5.      memset(swapper_pg_dir, 0x0, PGSIZE);  
6.  
7.      // No OpenSBI mapping required  
8.  
9.      // mapping kernel text X|-|R|V  
10.     create_mapping(swapper_pg_dir, (uint64_t)&_stext, (uint64_t)&  
        _stext - PA2VA_OFFSET, (uint64_t)&_srodata - (uint64_t)&_stext, 11);  
11.  
12.     // mapping kernel rodata -|-|R|V  
13.     create_mapping(swapper_pg_dir, (uint64_t)&_srodata, (uint64_t)  
        &_srodata - PA2VA_OFFSET, (uint64_t)&_sdata - (uint64_t)&_srodata, 3)  
        ;  
14.  
15.     // mapping other memory -|W|R|V
```

```

16.     create_mapping(swapper_pg_dir, (uint64_t)&_sdata, (uint64_t)&
    _sdata - PA2VA_OFFSET, PHY_SIZE - ((uint64_t)&_sdata - (uint64_t)&_s
    text), 7);
17.
18.     // set satp with swapper_pg_dir
19.
20.     // YOUR CODE HERE
21.     asm volatile (
22.         "mv t0, %[swapper_pg_dir]\n"
23.         ".set _pa2va_, 0xffffffffdf80000000\n"
24.         "li t1, _pa2va_\n"
25.         "sub t0, t0, t1\n"
26.         "srli t0, t0, 12\n"
27.         "addi t2, zero, 1\n"
28.         "li t1, 63\n"
29.         "sll t2, t2, t1\n"
30.         "or t0, t0, t2\n"
31.
32.         "csrw satp, t0\n"
33.
34.         : : [swapper_pg_dir] "r" (swapper_pg_dir)
35.         : "memory"
36.     );
37.
38.     // flush TLB
39.     asm volatile("sfence.vma zero, zero");
40.
41.     // flush icache
42.     asm volatile("fence.i");
43.     return;
44. }
45.
46.
47. /* 创建多级页表映射关系 */
48. /* 不要修改该接口的参数和返回值 */
49. void create_mapping(uint64_t *pgtbl, uint64_t va, uint64_t pa, ui
    nt64_t sz, uint64_t perm) {
50.     /*
51.      * pgtbl 为根页表的基地址
52.      * va, pa 为需要映射的虚拟地址、物理地址
53.      * sz 为映射的大小，单位为字节
54.      * perm 为映射的权限（即页表项的低 8 位）
55.      *
56.      * 创建多级页表的时候可以使用 kalloc() 来获取一页作为页表目录

```

```

57.      * 可以使用 v bit 来判断页表项是否存在
58.      **/
59.      uint64_t VPN[3];
60.      uint64_t *page_table[3];
61.      uint64_t new_page;
62.
63.      for (uint64_t address = va; address < va + sz; address += PGS
        IZE, pa += PGSIZE){
64.          page_table[2] = pgtbl;
65.          VPN[2] = (address >> 30) & 0x1ff;
66.          VPN[1] = (address >> 21) & 0x1ff;
67.          VPN[0] = (address >> 12) & 0x1ff;
68.          for (int level = 2; level > 0; level--){
69.              if ((page_table[level][VPN[level]] & 1) == 0){
70.                  new_page = kalloc();
71.                  page_table[level][VPN[level]] = (((new_page - PA2
        VA_OFFSET) >> 12) << 10) | 1;
72.              }
73.              page_table[level - 1] = (uint64_t*)(((page_table[leve
        l][VPN[level]] >> 10) << 12) + PA2VA_OFFSET);
74.          }
75.          page_table[0][VPN[0]] = ((pa >> 12) << 10) | (perm & 0x3f
        f);
76.      }
77.  }

```

最后，对 head.S 作调整，调整完后 head.S 如下：

```

1.      .extern start_kernel
2.      .extern setup_vm
3.      .extern setup_vm_final
4.      .extern mm_init
5.      .extern task_init
6.      .extern _traps
7.      .set PA2VA_OFFSET, 0xffffffffdf80000000
8.      .section .text.init
9.      .globl _start
10.     _start:
11.         la sp, boot_stack_top #store the address of stack top
12.         li t0, PA2VA_OFFSET
13.         sub sp, sp, t0
14.
15.         call setup_vm
16.         call relocate

```

```

17.    call mm_init
18.    call setup_vm_final
19.    call task_init
20.
21.    la t0, _traps
22.    csrw stvec, t0 #stvec = _traps
23.
24.    csrr t0, sie
25.    ori t0, t0, 0x20
26.    csrw sie, t0 #sie[STIE] = sie[5] = 1
27.
28.    andi a1, x0, 0
29.    andi a2, x0, 0
30.    andi a3, x0, 0
31.    andi a4, x0, 0
32.    andi a5, x0, 0
33.    andi a6, x0, 0
34.    andi a7, x0, 0
35.    li t0, 10000000
36.    rdttime a0
37.    add a0, a0, t0
38.    ecall #set first time interrupt
39.
40.    csrr t0, sstatus
41.    ori t0, t0, 0x2
42.    csrw sstatus, t0 #sstatus[SIE] = sstatus[1] = 1
43.
44.    jal start_kernel #jump to start_kernel
45.
46.    relocate:
47.    li t0, PA2VA_OFFSET
48.    add ra, ra, t0
49.    add sp, sp, t0
50.
51.    la t2, early_pgtbl
52.    sub t2, t2, t0
53.    srli t2, t2, 12
54.    addi t0, x0, 1
55.    li t1, 63
56.    sll t0, t0, t1
57.    or t2, t2, t0
58.
59.    sfence.vma zero, zero
60.    csrw satp, t2

```



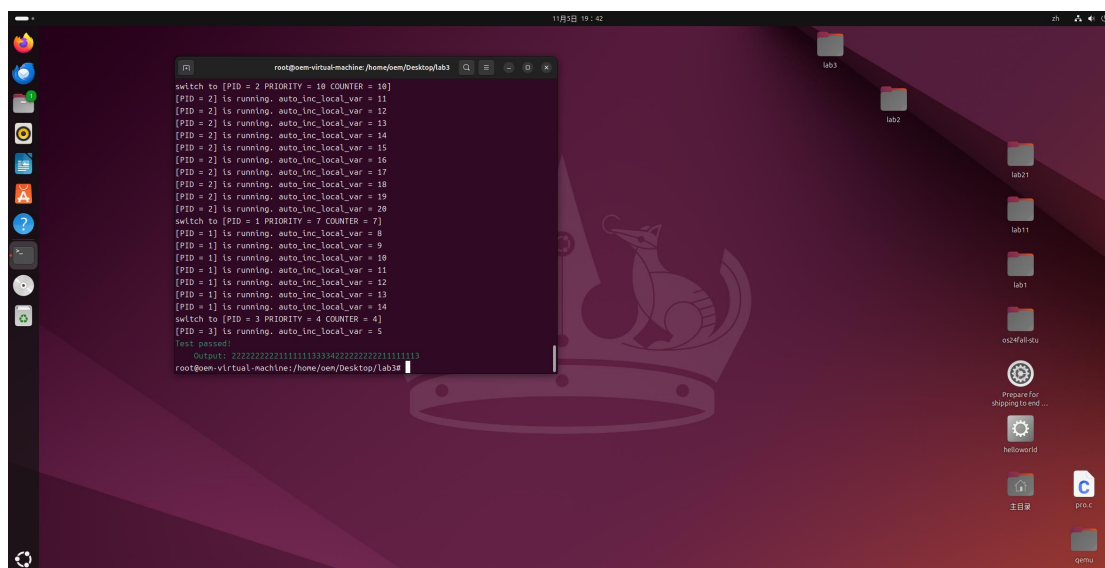
```

61.
62.
63.     ret
64.
65.     .section .bss.stack
66.     .globl boot_stack
67.     boot_stack:
68.     .space 4096 # <-- change to your stack size
69.
70.     .globl boot_stack_top
71.     boot_stack_top:

```

（三）编译及测试

为了验证算法正确性，本次实验加入了一个测试样例（在 4 个线程的情况下的 pid 输出）
测试结果如下：



```

root@qem-virtual-machine:/home/qem/Desktop/lab3
switch to [PID = 2 PRIORITY = 10 COUNTER = 10]
[PID = 2] is running. auto_inc_local_var = 11
[PID = 2] is running. auto_inc_local_var = 12
[PID = 2] is running. auto_inc_local_var = 13
[PID = 2] is running. auto_inc_local_var = 14
[PID = 2] is running. auto_inc_local_var = 15
[PID = 2] is running. auto_inc_local_var = 16
[PID = 2] is running. auto_inc_local_var = 17
[PID = 2] is running. auto_inc_local_var = 18
[PID = 2] is running. auto_inc_local_var = 19
[PID = 2] is running. auto_inc_local_var = 20
switch to [PID = 1 PRIORITY = 7 COUNTER = 7]
[PID = 1] is running. auto_inc_local_var = 8
[PID = 1] is running. auto_inc_local_var = 9
[PID = 1] is running. auto_inc_local_var = 10
[PID = 1] is running. auto_inc_local_var = 11
[PID = 1] is running. auto_inc_local_var = 12
[PID = 1] is running. auto_inc_local_var = 13
[PID = 1] is running. auto_inc_local_var = 14
switch to [PID = 3 PRIORITY = 4 COUNTER = 4]
[PID = 3] is running. auto_inc_local_var = 5
Test passed
Output: 22222222211111113333442222222211111119
root@qem-virtual-machine:/home/qem/Desktop/lab3

```

根据图中显示，测试通过

三、讨论和心得

本次实验建立在 lab2 环境已经搭建好的基础上，做起来并不是特别困难，但是需要理解虚拟内存的应用，特别是需要关注内核如何开启虚拟地址以及通过设置页表来实现地址映

射和权限控制。

这次实验让我对虚拟内存的工作机制有了更深入的了解，同时也知道了 `satp` 寄存器的使用。

四、思考题

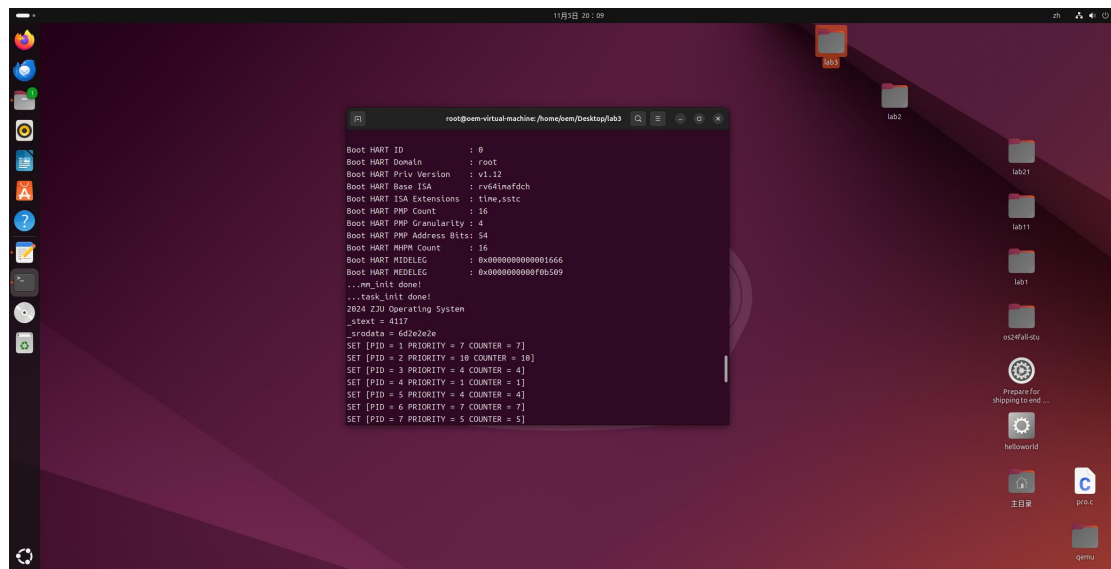
1.

编译成功后能够正常运行，说明 `.text` 段具有执行权限，`.rodata` 段不具有执行权限。

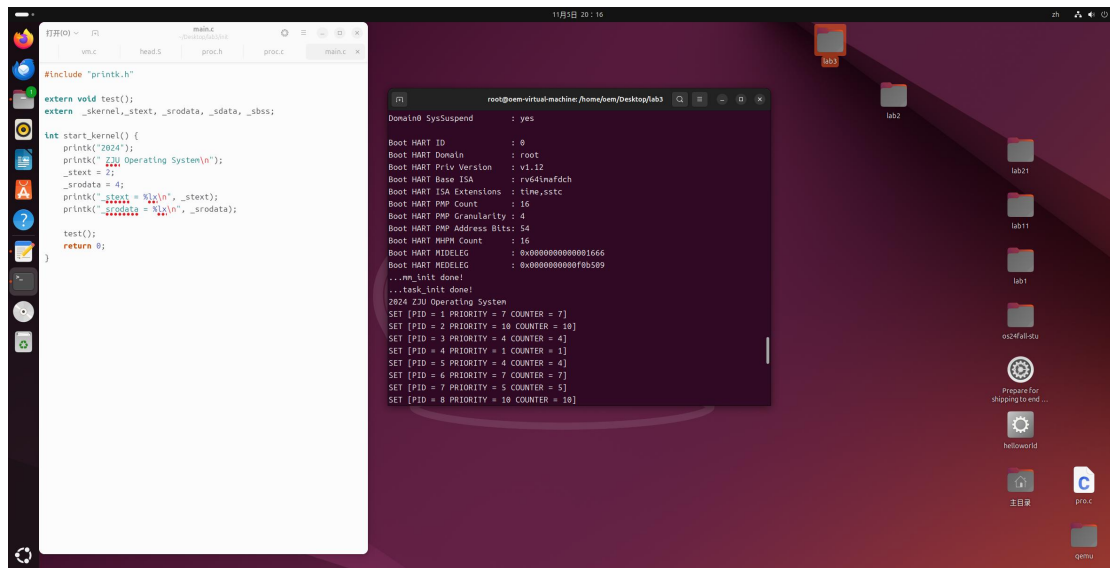
在 `main.c` 中，声明外部变量 `_stext`，`_srodata`，在 `start_kernel` 中加入以下代码，读取 `.text` 段和 `.rodata` 段的数据并输出：

```
1. printk("_stext = %lx\n", _stext);
2. printk("_srodata = %lx\n", _srodata);
```

编译运行后，发现终端正常输出数据，说明均有读取权限：



而当添加写入的代码时，无法输出 `_stext` 和 `_srodata` 的值，说明这两端均无写入权限：



2.

本次实验中如果不做等值映射，会出现什么问题，原因是什么？

开启虚拟地址后，不能直接访问物理地址。在三级映射中，页表地址事实上都是物理地址，如果不做等值映射，那么虚拟内存中不存在这一段地址，就无法访问页表。

简要分析 Linux v5.2.21 或之后的版本中的内核启动部分（直至 `init/main.c` 中 `start_kernel` 开始之前），特别是设置 `satp` 切换页表附近的逻辑。

首先通过 `vmlinux` 文件的链接脚本（`vmlinux.lds`），从中查找系统启动入口函数，得到内核入口函数。接着调用 `safe_svcmode_maskal()` 进入 SVC 模式，并关闭中断；读取处理器 ID，保持在 `r9` 寄存器；调用 `__create_page_tables` 创建页表，经历了两次创建页表过程，第一次使用 `setup_vm()` 创建临时页表，通过 `trampoline_pg_dir` 将虚拟内存的地址 `PAGE_OFFSET` 映射到物理地址 `load_pa`，再通过 `early_pg_dir` 将从 `PAGE_OFFSET` 到 `end_va` 的虚拟地址映射到对应的物理地址，然后在 `relocate` 中先将 `satp` 寄存器设置为 `trampoline_pg_dir` 的地址，然后进入异常 1f 段，重新设置异常入口为 `Lsecondary_park`，切换到 `early_pg_dir` 页表，接着调用 `setup_vm_final()` 函数，使用 `swapper_pg_dir` 页表管理整个物理内存的访问，把 `swapper_pa_dir` 的物理地址赋值给 `satp` 寄存器；然后调用 `__enable_mmu` 函数使能 MMU；最后启动 `__mmap_switched`。

Linux 为什么可以不进行等值映射，它是在如何在无等值映射的情况下让 `pc` 从物理地址跳到虚拟地址？

在内核启动时，内核会使用 `trampoline_pg_dir` 页表来实现从虚拟地址到物理地址的映射，同时 `early_pg_dir` 会建立 `fixmap_pgd_next`, `fixmap_pte`, `early_dtb_pmd` 页表虚拟地址到物理地址的映射。在 `relocate` 函数中，计算返回地址 `ra = ra + 物理地址到虚拟地址的偏移`，使得程序能够返回虚拟地址，在最终页表的建立过程中，根页表以及中间级别的页表都使用了虚拟地址，因此 linux 不需要直接访问物理地址，也就是可以不进行等值映射。

Linux v5.2.21 中的 `trampoline_pg_dir` 和 `swapper_pg_dir` 有什么区别，它们分别是在哪里通过 `satp` 设为所使用的页表的？

trampoline_pg_dir 用于设置虚拟内存，帮助内核从物理地址跳转到虚拟地址。swapper_pg_dir 负责将内核的虚拟地址空间映射到物理地址空间，包含内核代码、数据、堆栈等的映射。trampoline_pg_dir 在 relocate 函数通过 satp 设为所使用的页表，swapper_pg_dir 在 setup_vm_final() 中把值写入 satp 寄存器。

尝试修改你的 kernel，使得其可以像 Linux 一样不需要等值映射。

将页表项中读取到的页号计算得到的物理地址转换为虚拟地址，供下一次访问使用。

```
1.    page_table[level - 1] = (uint64_t*)((page_table[level][VPN[level]]
    ] >> 10) << 12) + PA2VA_OFFSET);
```

五、附录

无